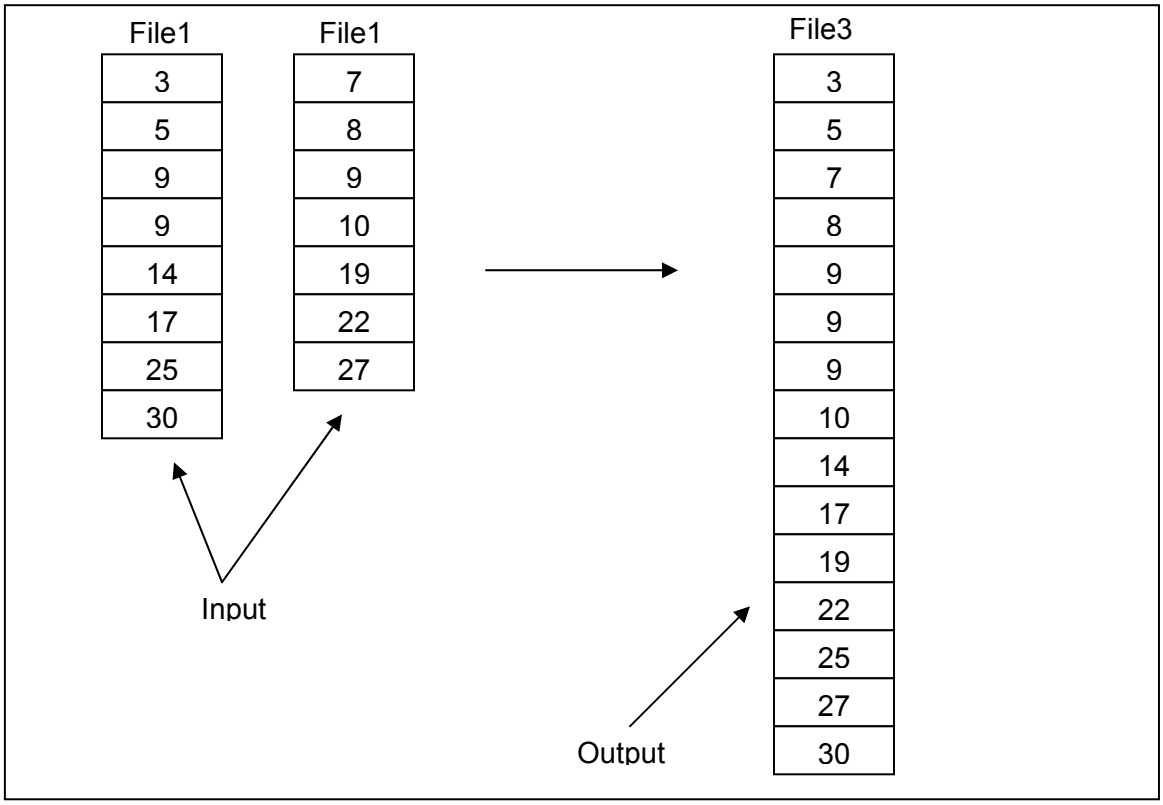
Bilkent University
Computer Engineering Department

CS 342, Operating Systems
Project 3
Parallel Merge

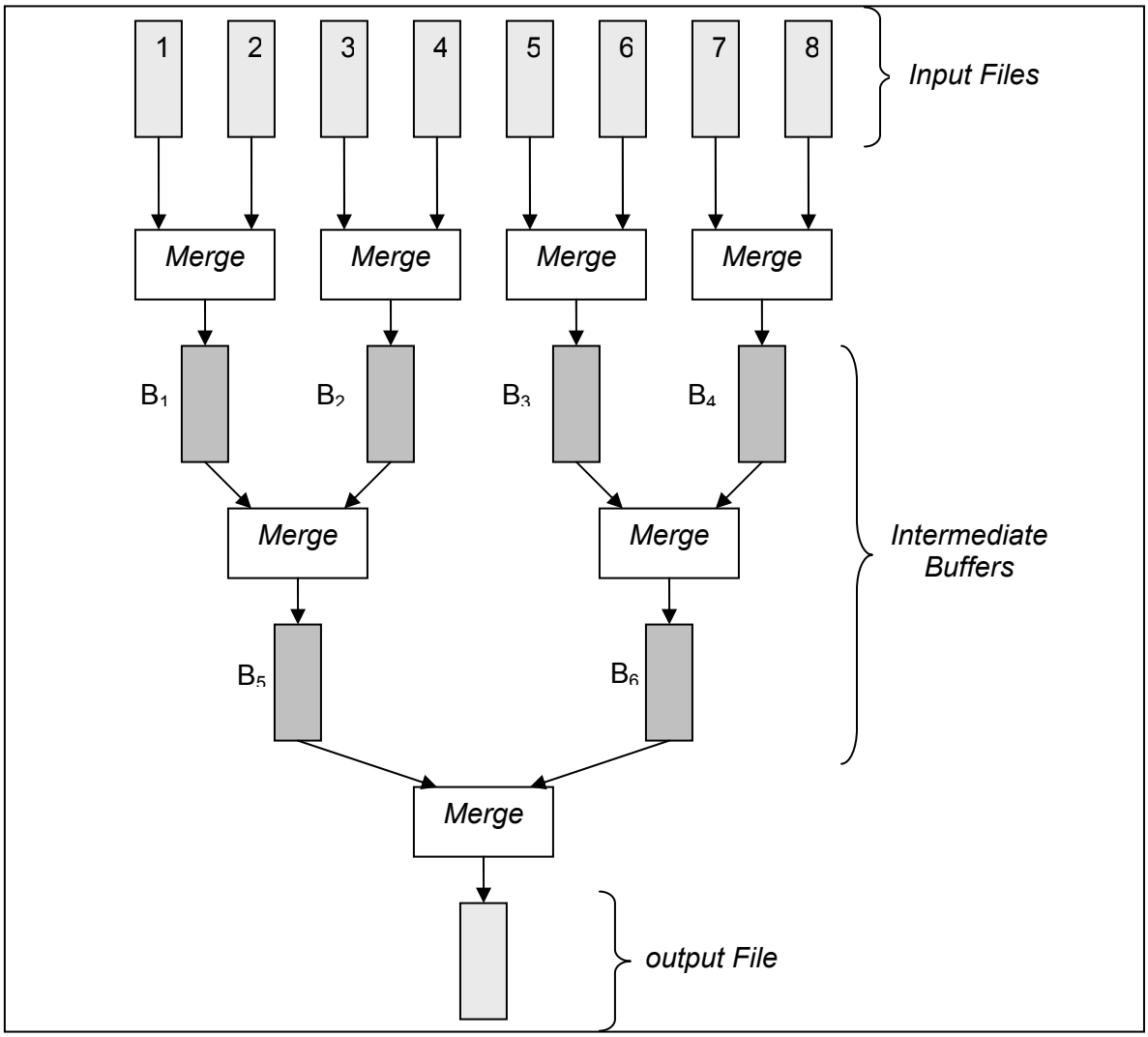
- You will implement this project using Linux Operating System
- You can implement it in an other system, but before submission you have to port your implementation to Linux, since Linux will be the platform on which we will perform our tests.
- The project will be done individually.
- You will be required to do demos of your programs after you submit. In the demo, you will be asked questions, will be required to compile and run your programs, and make modifications if required by the instructor and TAs.
- The project will be done in C programming language.

In this project you will implement a *parallel merge* of sorted integers stored in N input files. An input file will be an ascii file containing sorted integers in ascending order. Each line of an input file will contain exactly one integer. For example merging two input files into one output file will be like the following:



The number input files will not be only two, but can be a power of two (4,8,16, etc.). In that case, you will merge the files in several phases. A merge will be done always

between two files (no more than that). The idea is shown in the figure below. There are eight input files, each of which stores sorted integers. Then these files will be merged into four intermediate buffers. At the same time (concurrently), the content of these buffers will merged into two other intermediate buffers. Again at the same time, the content of these two other buffers will be merged into a final output file. Note that the buffers B_1 through B_6 are shared buffers. While one merge operation is using a buffer for output, an other merge operation is using that buffer for input. This is similar to producer-consumer problem that we have seen in the class.



You will implement and realize this idea by using two approaches:

1. Use of processes to implement the *merge* operations (*merge boxes* in the figure above) and use of shared memory to implement the *shared intermediate buffers* which we have described above.

2. Use of threads (that are spawned from a single process) to implement the merge operations, and use of thread global variables (data) to implement the *shared intermediate buffers.*

First Approach – Concurrent processes

In this approach, you will implement a parent process that will invoke some number of children that is enough to implement the idea above. For example, in the figure above, there will be a total of 7 child processes each of which implement one merge operation. One merge operation can read the input from two separate files or from two intermediate buffers. The output can be written to an intermediate buffer or to a final output file. You will create the children using the `fork()` system call. The invocation of your parent process will be as the following:

```
parallelmerge1 −n 4 file1 file2 file3 file4 -o outputfile
```

The `parallelmerge1` is the name of the program. There are four input files (number of input files need to be power of two). The `−n` option is used to specify the number of input files. The final output will be written to file outputfile that is specified using `−o` option. The outputfile will contain exactly one integer at each line.

The processes will communicate the integers with each other using shared memory facility that is provided in most Unix systems. You will implement the shared buffers using this shared memory facility.

You will use *semaphores* to provide mutual exclusion and synchronization while multiple processes are accessing a shared buffer that is sitting in the shared memory area.

Second Approach – Concurrent Threads.

In the second approach, you will use concurrent threads to implement the merge operations. For example in the figure above, there would be seven threads that are created by a main thread (or call it main process). These seven threads are running concurrently and each of these threads is doing one merge operation as shown in the figure. A thread, while doing merge, will take the input from either two files or two shared buffers. A thread will write the output to either a final output file or to an other shared buffer. The threads will be created using the `thread_create()` function (or the corresponding function in the Unix system that you are using). The invocation of your main process (that will create the threads) will be as follows:

```
parallelmerge2 −n 4 file1 file2 file3 file4 -o outputfile
```

The `parallelmerge2` is the name of the main program. The `−n` option specifies the number of input files. Then comes the name of the input files. The name of the output file is specified using the `−o` option.

The shared buffers will be implemented as global structures in the main process, so that all threads can share and access them. Since multiple threads will be accessing and modifying a shared buffer, you should use *mutexes* and *condition variables* to provide mutual exclusion and synchronization. Otherwise, your results may be inconsistent and wrong.

You will get the *submission rules* later.